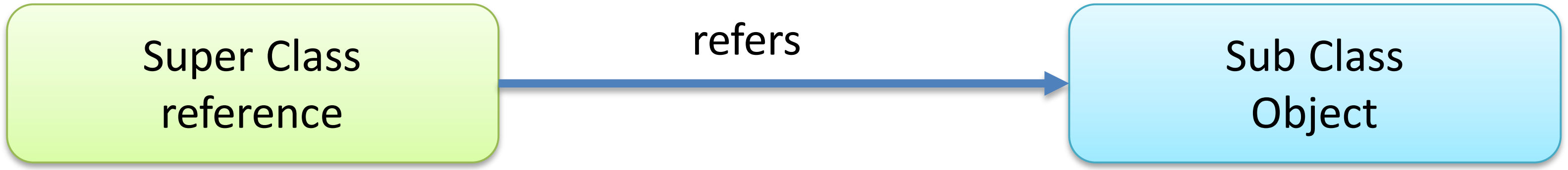
# Object Oriented Programming with Java 8

Dr.Akshita Chanchlani

# Upcasting

When the reference variable of super class refers to the object of subclass, it is known as widening or **upcasting in java**.

when subclass object type is converted into superclass type, it is called widening or upcasting.



Superclass s = new SubClass();

***Up casting :*** *Assigning child class object to parent class reference .*
Syntax for up casting : **Parent p = new Child();**
Here **p** is a parent class reference but point to the child object. *This reference p can access all the methods and variables of parent class but only overridden methods in child class.*

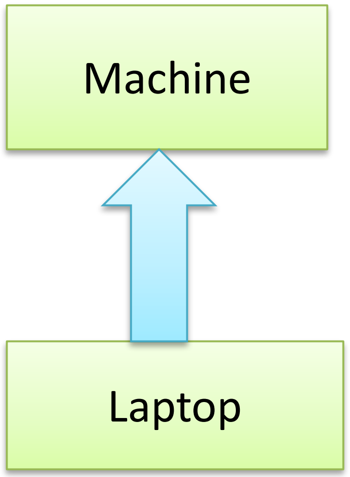
Upcasting gives us the flexibility to access the parent class members, but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can only access the overridden methods in the child class.

# Downcasting

***Down casting :*** *Assigning parent class reference (which is pointing to child class object) to child class reference .*

Syntax for down casting : **Child c = (Child)p;**

Here **p** is pointing to the object of child class as we saw earlier and now we cast this parent reference **p** to child class reference **c**. *Now this child class reference **c** can access all the methods and variables of child class as well as parent class.*



*For example, if we have two classes, **Machine** and **Laptop** which extends **Machine** class. Now for upcasting, every laptop will be a machine but for downcasting, every machine may not be a laptop because there may be some machines which can be **Printer**, **Mobile,** etc.*
**Downcasting is not always safe, and we explicitly write the class names before doing downcasting.** So that it won't give an error at compile time but it may throw **ClassCastExcpetion** at run time, if the parent class reference is not pointing to the appropriate child class.

## Explanation

```
Machine machine = new Machine ();

Laptop laptop = (Laptop)machine;//this won't give an error while compiling

//laptop is a reference of type Laptop and machine is a reference of type Machine and points to Machine
class Object .So logically assigning machine to laptop is invalid because these two classes have different
object strucure.And hence throws ClassCastException at run time .
```

**To remove ClassCastException we can use instanceof operator to check right type of class reference in case of down casting .**

```
if(machine instanceof Laptop)
{
Laptop laptop = machine;      //here machine must be pointing to Laptop class object .
}
```

## Polymorphisim

- The ability to have many different forms
- An object always has only one form
- A reference variable can refer to objects of different forms

# Method Binding

## Static Binding

- Static binding
- Compile time
- early binding
- Resolved by java compiler
- Achieved via method overloading

Example :

In class Test :

```
void test(int i,int j){...}
void test(int i) {..}
void test(double i){..}
void test(int i,double j,boolean flag){..}
int test(int a,int b){...}   //javac error
```

## Dynamic Binding

- Dynamic binding
- Run time / Late binding
- Resolved by java runtime environment
- Achieved by method overriding (Dynamic method dispatch)
- Method Overriding is a Means of achieving run-time polymorphism

All java methods can be overridden : if they are not marked as private or static or final

Super-class form of method is called as overridden method

sub-class form of method is called as overriding form of the method

Example :

```
class A {
   A getInstance()
             {
               return new A();
             }
   }
```

```
class B extends A
{
   B getInstance()
             {
               return new B();
             }
}
```

# Run time polymorphism or Dynamic method dispatch

- Super -class ref. can directly refer to sub-class object(direct=w/o type casting) as its the example of up-casting(similar to widening auto. conversion) .

- When such a super class ref is used to invoke the overriding method then the method to send for execution that decision is taken by JRE & not by compiler.
- In such case overriding form of the method(sub-class version) will be dispatched for exec.
- Javac resolves the method binding by the type of the reference & JVM resolves the method binding by type of the object it's referring to.
- Super class reference can directly refer to sub-class instance BUT it can only access the members declared in super-class directly.
- eg : A ref=new B();

  ref.show(); // this will invoke the sub-class: overriding form of the show () method

Java compiler resolves method binding by type of the reference & JVM resolves it by the type of the object, reference is referring to.

## Annotation

- From JDK 1.5 onwards : Annoations are available , it is metadata meant for Compiler or JRE.(Java tools)

- Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

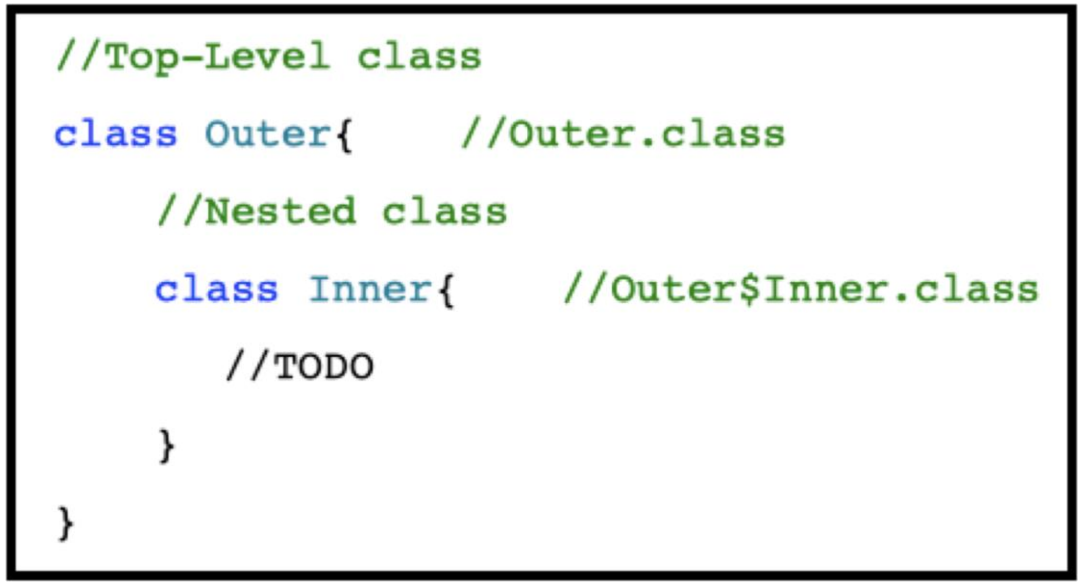
- Annotations in java are used to provide additional information, so it is an alternative option for XML.

- eg @Override,@Deprecated,@SuppressWarnings,@FunctionalInterface

- @Override
- It is an annotation meant  for javac.
- It's Method level annotation ,that appears in a sub class
- It's Optional BUT recommended.
- While overriding the method if you want to inform the compiler that : following is the overriding form of the method use :
- @Override
- method declaration {...}

# Nested Class

- In Java, we can define class inside scope of another class. It is called nested class.
- Nested class represents encapsulation.

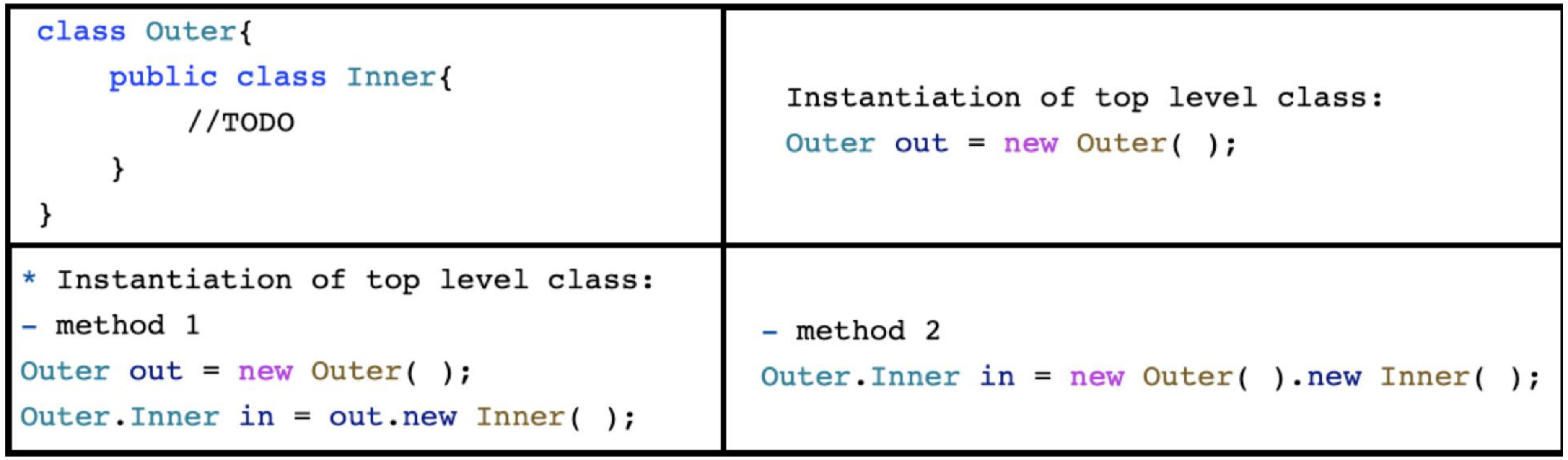


```
//Top-Level class
class Outer{    //Outer.class
    //Nested class
    class Inner{    //Outer$Inner.class
       //TODO
    }
}
```

- Access modifier of top level class can be either package level private or public only.
- We can use any access modifier on nested class.
- Types of nested class:
    1. Non static nested class / Inner class
    2. Static nested class

# Non Static Nested Class

- Non static nested class is also called as inner class.
- If implementation of nested class depends on implementation of top level class then nested class should be non static.
- **Implementation Hint** : For the simplicity, consider non static nested class as non static method of class.



```
class Outer{
    public class Inner{
        //TODO
    }
}
```

```
Instantiation of top level class:
Outer out = new Outer( );
```

```
* Instantiation of top level class:
- method 1
Outer out = new Outer( );
Outer.Inner in = out.new Inner( );
```

```
- method 2
Outer.Inner in = new Outer( ).new Inner( );
```

# Non Static Nested Class

- Top level class can contain static as well as non static members.
- Inside non static nested class we can not declare static members.
- If we want to declare any field static then it must be final.
- Using instance, we can access members of non static nested class inside method of top level class.
- Without instance, we can use all the members of top level class inside method of non static nested class.
- But if we want refer member of top level class, inside method of non static nested class then we should use "TopLevelClassName.this" syntax.

# Non Static Nested Class

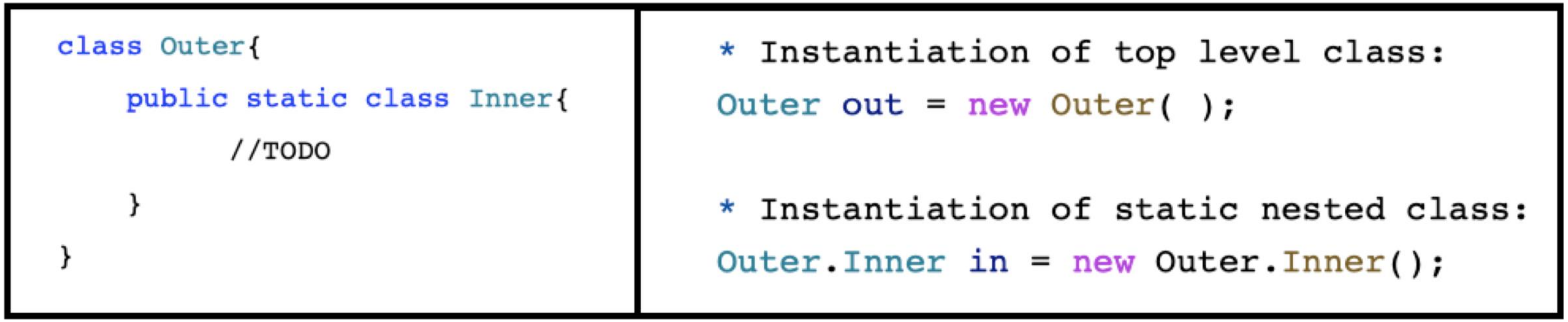
```java
class Outer{
    private int num1 = 10;

    public class Inner{
        private int num1 = 20;

        public void print( ) {
            int num1 = 30;
            System.out.println("Num1     :    "+Outer.this.num1); //10
            System.out.println("Num1     :    "+this.num1);   //20
            System.out.println("Num1     :    "+num1);    //30
        }
    }
}
public class Program {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.print();
    }
}
```

## Static Nested Class

- If we declare nested class static then it is simply called as static nested class.
- We can declare nested class static but we can not declare top level class static.
- If implementation of nested class do not depends on implementation of top level class then we should declare nested class static.
- **Implementation hint:** For simplicity consider static nested class as a static method of a class.



```
class Outer{
    public static class Inner{
        //TODO
    }
}
```

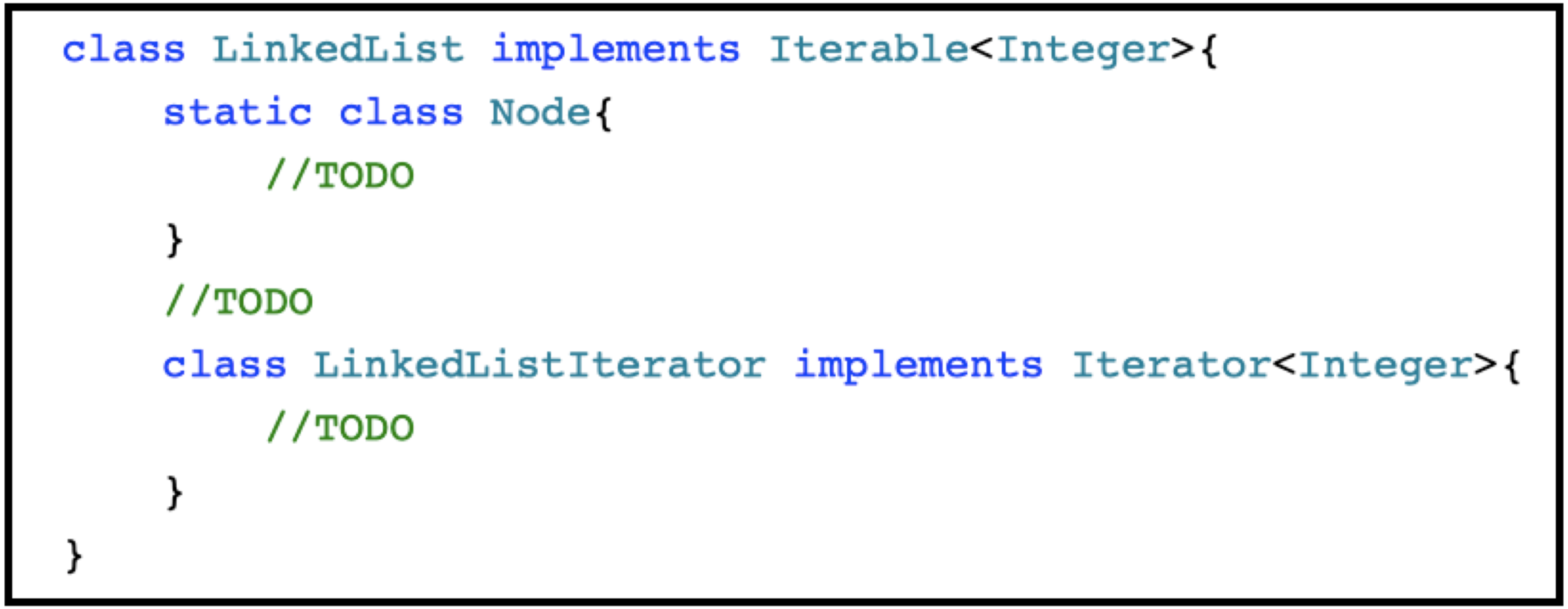
```
* Instantiation of top level class:
Outer out = new Outer( );

* Instantiation of static nested class:
Outer.Inner in = new Outer.Inner();
```

## Static Nested Class

- Static nested class can contain static members.
- Using instance, we can access all the members of static nested class inside method of top level class.
- If we want to use non static members of top level class inside method of static nested class then it is mandatory to create instance of top level class.

## Nested Class



```
class LinkedList implements Iterable<Integer>{
    static class Node{
        //TODO
    }
    //TODO
    class LinkedListIterator implements Iterator<Integer>{
        //TODO
    }
}
```

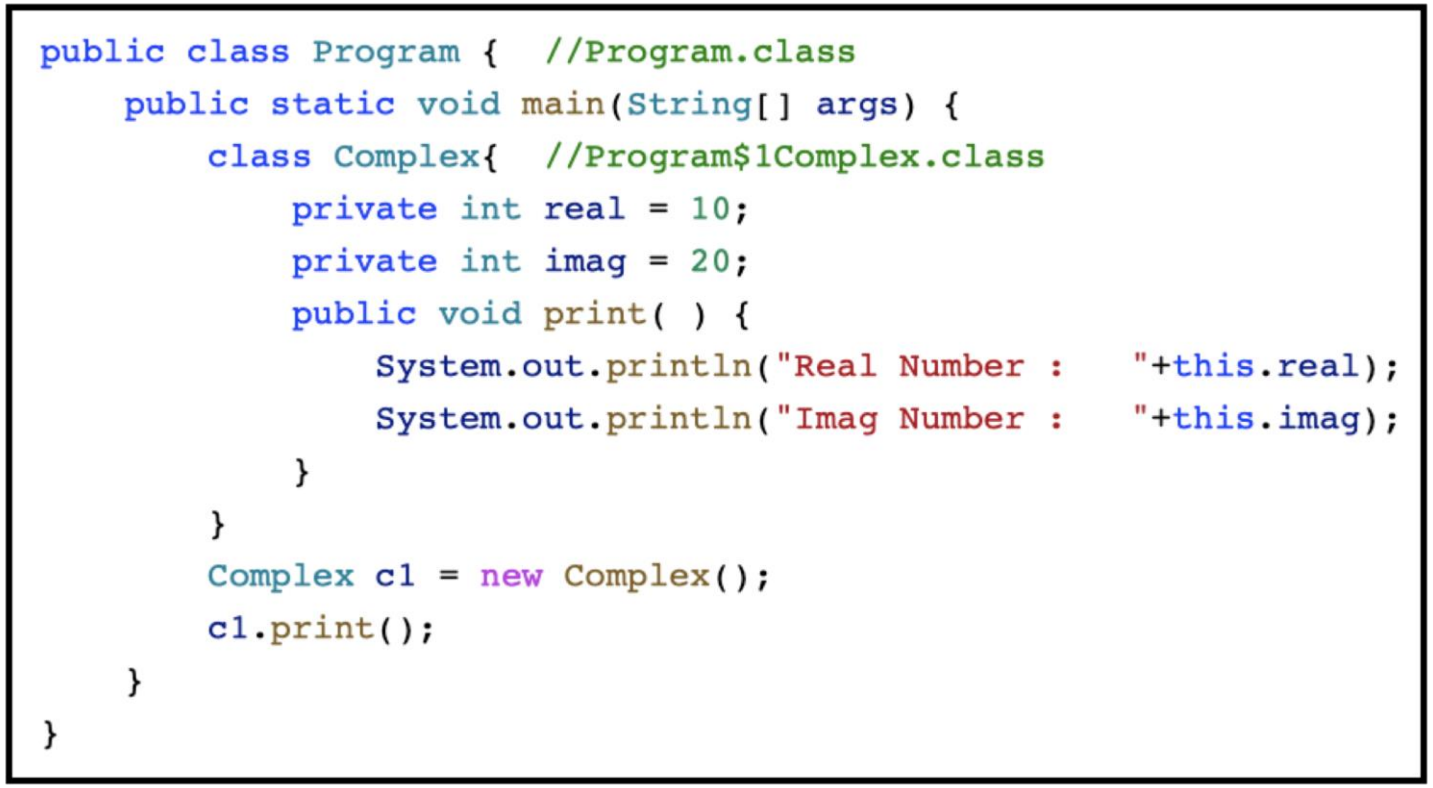
## Local Class

- In Java, we can define class inside scope of another method. It is called local class / method local class.

- Types of local class:
    1. Method local inner class
    2. Method local anonymous inner class.

# Method Local Inner Class

- In Java, we can not declare local variable /class static hence local class is also called as local inner class.
- We can not use reference/instance of method local class outside method.



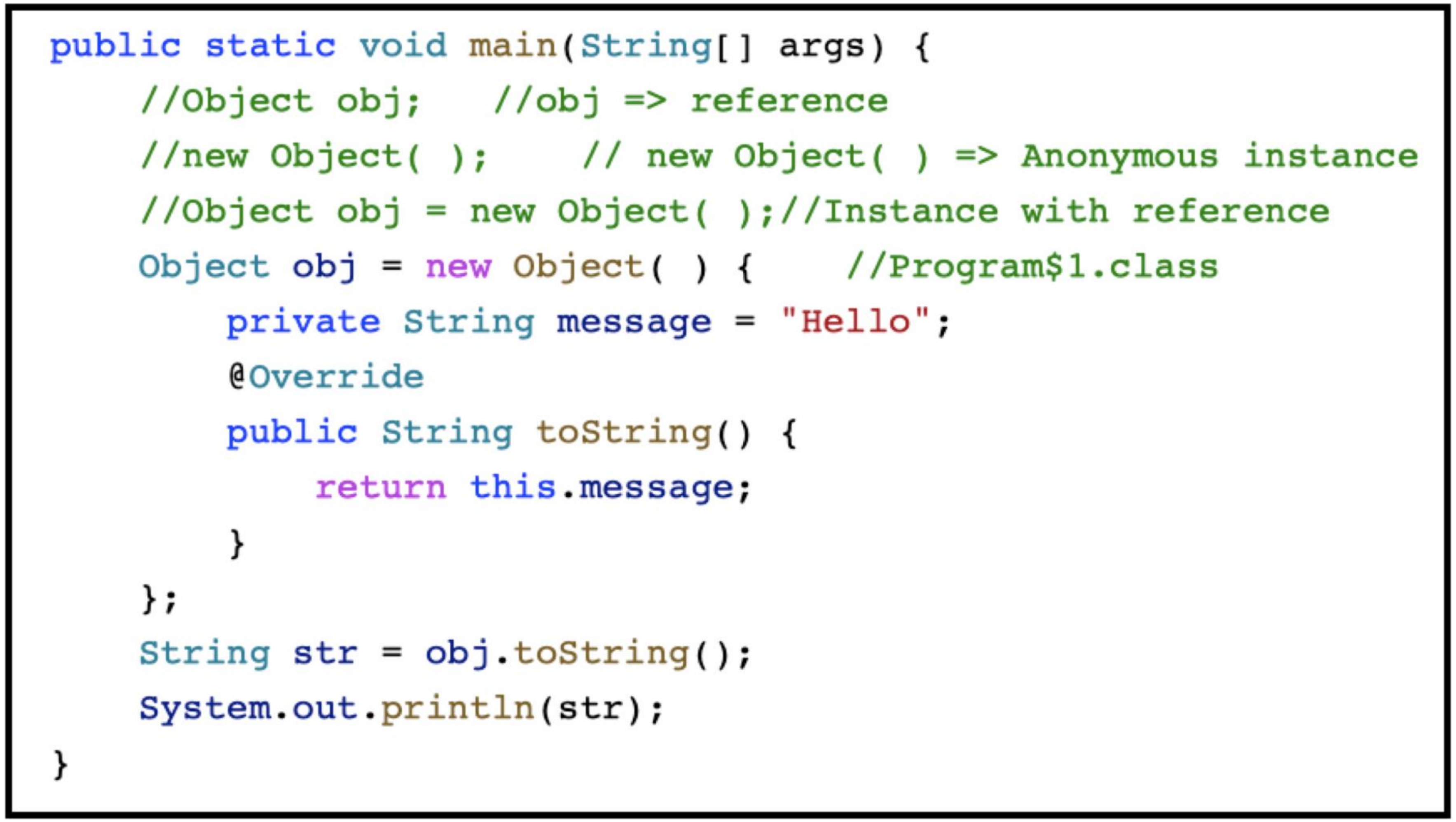
```
public class Program {  //Program.class
    public static void main(String[] args) {
        class Complex{  //Program$1Complex.class
            private int real = 10;
            private int imag = 20;
            public void print( ) {
                System.out.println("Real Number :    "+this.real);
                System.out.println("Imag Number :    "+this.imag);
            }
        }
        Complex c1 = new Complex();
        c1.print();
    }
}
```

## Method local anonymous inner class.

- In java, we can create instance without reference. It is called anonymous instance.
- Example:
- `new Object( );`
- We can define a class without name. It is called anonymous class.
- If we want to define anonymous class then we should use new operator.
- We can create anonymous class inside method only hence it is also called as method local anonymous class.
- We can not declare local class static hence it is also called as method local anonymous inner class.
- To define anonymous class, we need to take help of existing interface / abstract class / concrete class.

## Method local anonymous inner class.

- Consider anonymous inner class using concrete class.



```
public static void main(String[] args) {
    //Object obj;   //obj => reference
    //new Object( );    // new Object( ) => Anonymous instance
    //Object obj = new Object( );//Instance with reference
    Object obj = new Object( ) {    //Program$1.class
        private String message = "Hello";
        @Override
        public String toString() {
            return this.message;
        }
    };
    String str = obj.toString();
    System.out.println(str);
}
```

## Method local anonymous inner class.

- Consider anonymous inner class using abstract class:

```
abstract class Shape{
    protected double area;
    public abstract void calculateArea( );
    public double getArea() {
        return area;
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        Shape sh = new Shape() {
            private double radius = 10;
            @Override
            public void calculateArea() {
                this.area = Math.PI * Math.pow(this.radius, 2);
            }
        };

        sh.calculateArea();
        System.out.println("Area    :   "+sh.getArea());
    }
}
```

## Method local anonymous inner class.

- Consider anonymous inner class using interface.

```java
interface Printable{
    void print( );
}
```

```java
public class Program {
    public static void main(String[] args) {
        Printable p = new Printable() {
            @Override
            public void print() {
                System.out.println("Hello");
            }
        };
        p.print();
    }
}
```

## Interface

## Interface

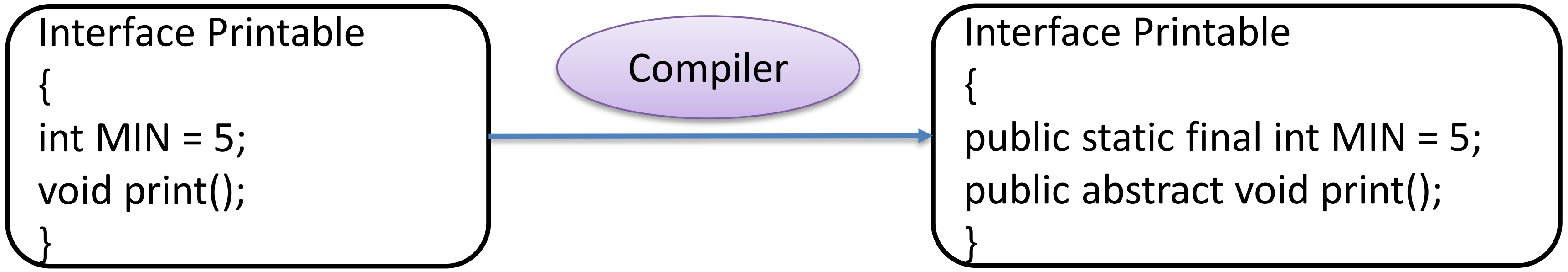
- In Java, an **interface** is a blueprint or template of a class. It is much similar to the Java class but the only difference is that it has abstract methods and static constants.
- An interface provides specifications of what a class should do or not and how it should do. An interface in Java basically has a set of methods that class may or may not apply.
- It also has capabilities to perform a function. The methods in interfaces do not contain any body.
- An interface in Java is a mechanism which we mainly use to achieve abstraction and multiple inheritances in Java.
- An interface provides a set of specifications that other classes must implement.
- We can implement multiple Java Interfaces by a Java class. All methods of an interface are implicitly public and abstract. The word abstract means these methods have no method body, only method signature.
- Java Interface also represents the IS-A relationship of inheritance between two classes.
- An interface can inherit or extend multiple interfaces.
- We can implement more than one interface in our class.

## Interface Vs Class

- Unlike a class, you cannot instantiate or create an object of an interface.
- All the methods in an interface should be declared as abstract.
- An interface does not contain any constructors, but a class can.
- An interface cannot contain instance fields. It can only contain the fields that are declared as both static and final.
- An interface can not be extended or inherited by a class; it is implemented by a class.
- An interface cannot implement any class or another interface.

## Syntax Interface

```
interface interface-name
{
//abstract methods
}
```



```
Interface Printable
{
int MIN = 5;
void print();
}
```

```
Interface Printable
{
public static final int MIN = 5;
public abstract void print();
}
```

## Interface

- Set of rules are called specification/standard.
- It is a contract between service consumer and service provider.
- If we want to define specification for the sub classes then we should define interface.
- Interface is non primitive type which helps developer:
  1. To build/develop trust between service provider and service consumer.
  2. To minimize vendor dependency.
- interface is a keyword in Java.

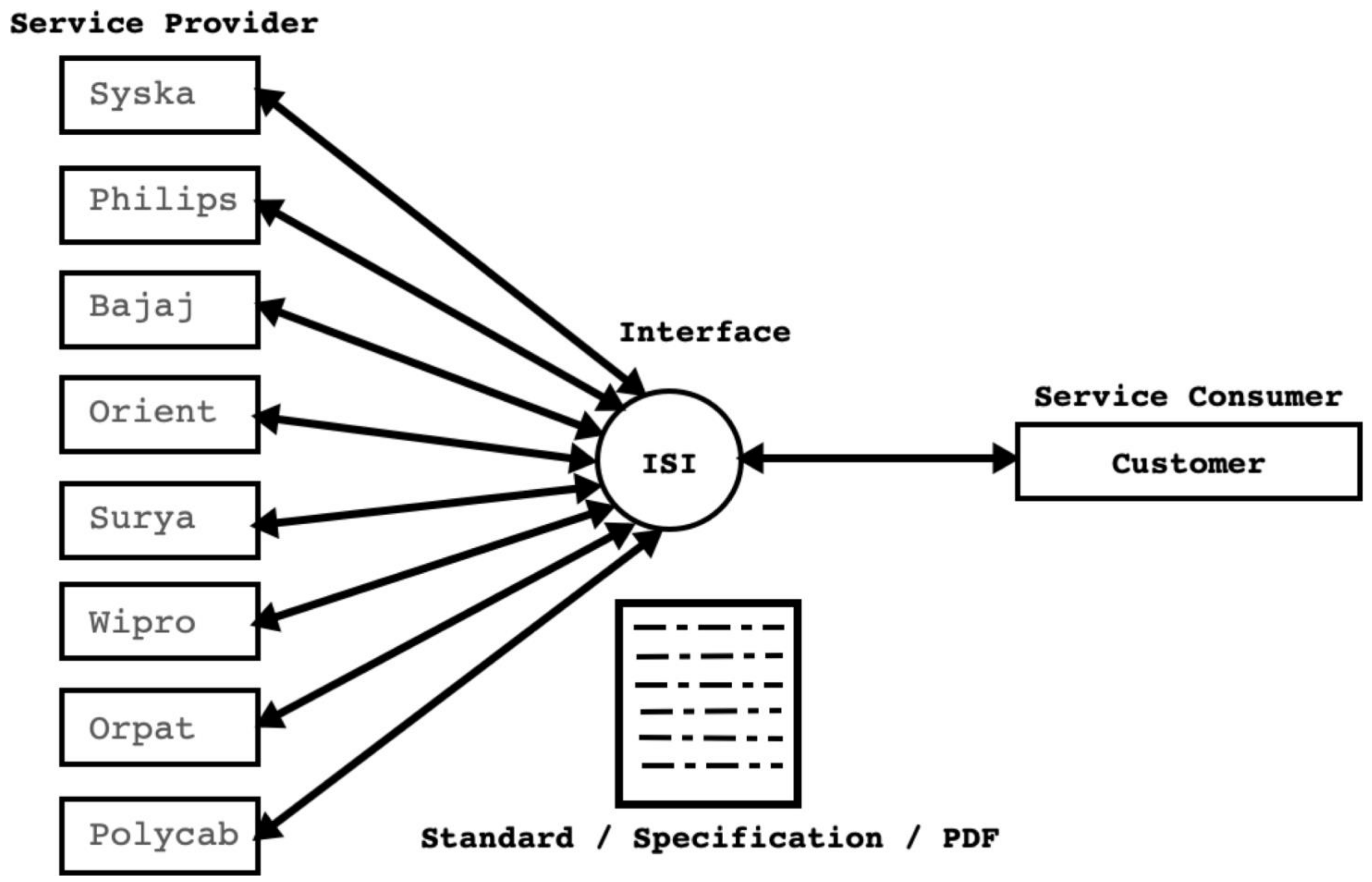
```
interface Printable{
    //TODO
}
```

# Interface

- Interface can contain:
    1. Nested interface
    2. Field
    3. Abstract method
    4. Default method
    5. Static method
- Interfaces cannot have constructors.
- We can create reference of interface but we can not create instance of interface.
- We can declare fields inside interface. Interface fields are by default public static and final.
- We can write methods inside interface. Interface methods are by default considered as public and abstract.

```
interface Printable{
    int number = 10; //public static final int number = 10;
    void print( ) ; //public abstract void print( ) ;
}
```

# Interface


Service Provider
Syska
Philips
Bajaj
Orient
Surya
Wipro
Orpat
Polycab
Interface
ISI
Service Consumer
Customer
Standard / Specification / PDF

## Interface

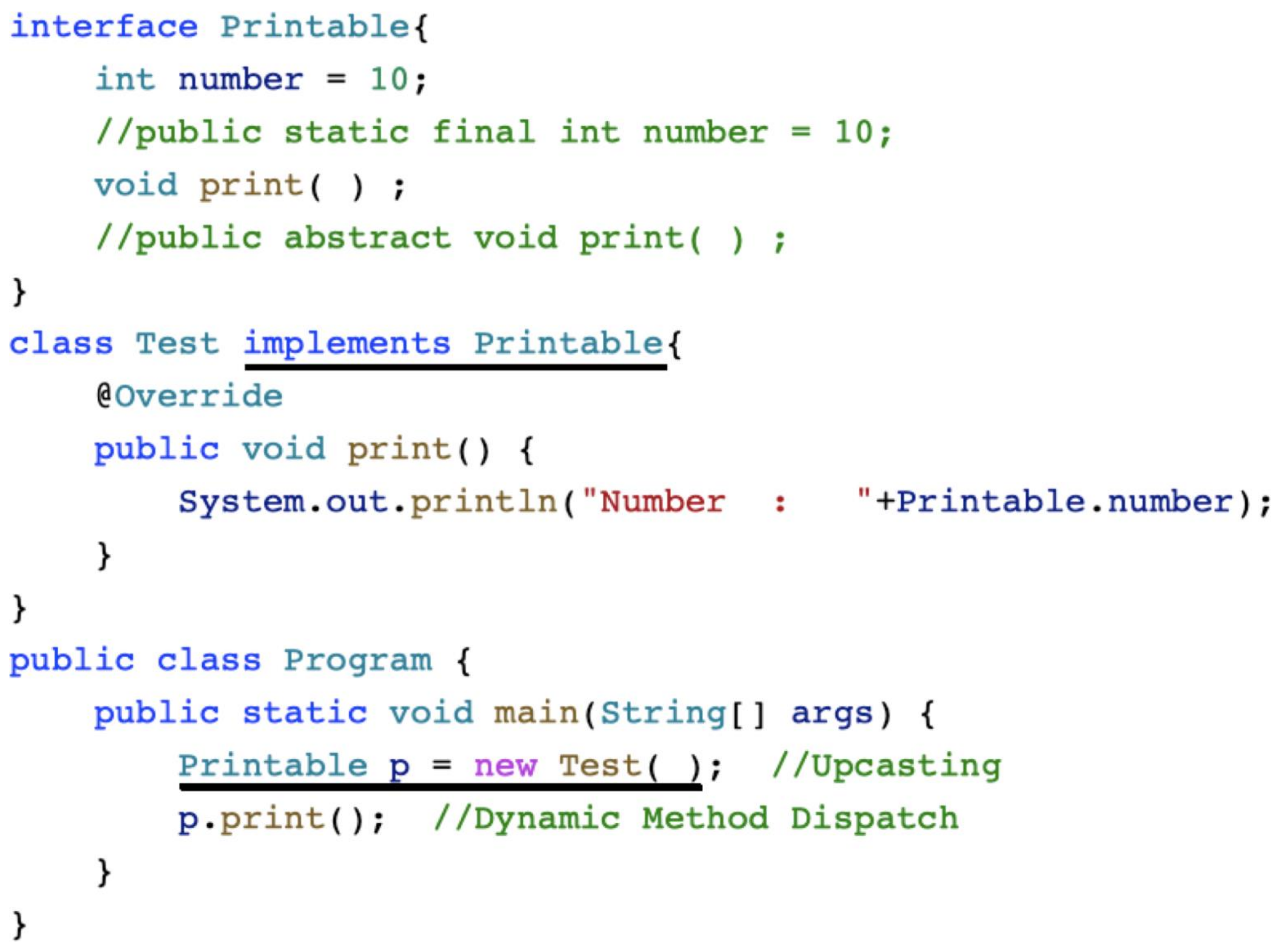
- If we want to implement rules of interface then we should use implements keyword.
- It is mandatory to override, all the abstract methods of interface otherwise sub class can be considered as abstract.

```
interface Printable{
    int number = 10;
    void print( );
}
```

```
* Solution 1
abstract class Test implements Printable{
}
```

```
* Solution 2
class Test implements Printable{
    @Override
    public void print( ){
        //TODO
    }
}
```

# Interface Implementation Inheritance



```java
interface Printable{
    int number = 10;
    //public static final int number = 10;
    void print( ) ;
    //public abstract void print( ) ;
}
class Test implements Printable{
    @Override
    public void print() {
        System.out.println("Number   :   "+Printable.number);
    }
}
public class Program {
    public static void main(String[] args) {
        Printable p = new Test( );  //Upcasting
        p.print();  //Dynamic Method Dispatch
    }
}
```

# Interface Syntax

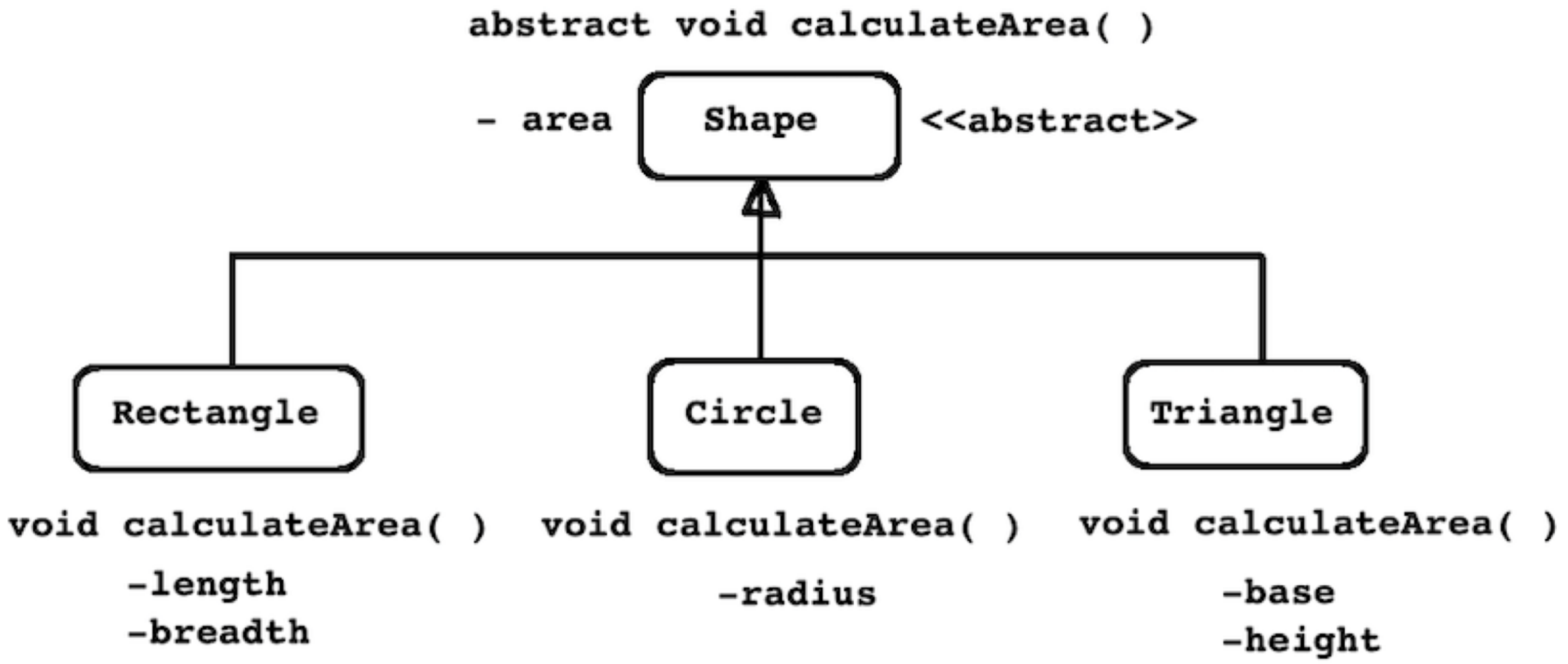
```
Interface : I1, I2, I3
Class     : C1, C2, C3

* I2 implements I1              //Incorrect
* I2 extends I1                 //correct : Interface inheritance
* I3 extends I1, I2             //correct : Multiple interface inheritance
* C2 implements C1              //Incorrect
* C2 extends C1                 //correct : Implementation Inheritance
* C3 extends C1,C2              //Incorrect : Multiple Implementation Inheritance
* I1 extends C1                 //Incorrect
* I1 implements C1              //Incorrect
* c1 implements I1              //correct : Interface implementation inheritance
* c1 implements I1,I2 //correct : Multiple Interface implementation inheritance
* c2 implements I1,I2 extends C1      //Incorrect
* c2 extends C1 implements I1,I2      //correct
```

## Types of inheritance

- **Interface Inheritance**
  - During inheritance if super type and sub type is interface then it is called interface inheritance.
    1. Single Inheritance( Valid in Java)
    2. Multiple Inheritance( Valid in Java)
    3. Hierarchical Inheritance( Valid in Java)
    4. Multilevel Inheritance( Valid in Java)
- **Implementation Inheritance**
  - During inheritance if super type and sub type is class then it is called implementation inheritance.
    1. Single Inheritance( Valid in Java)
    2. Multiple Inheritance( Invalid in Java)
    3. Hierarchical Inheritance( Valid in Java)
    4. Multilevel Inheritance( Valid in Java)

# Abstract Class



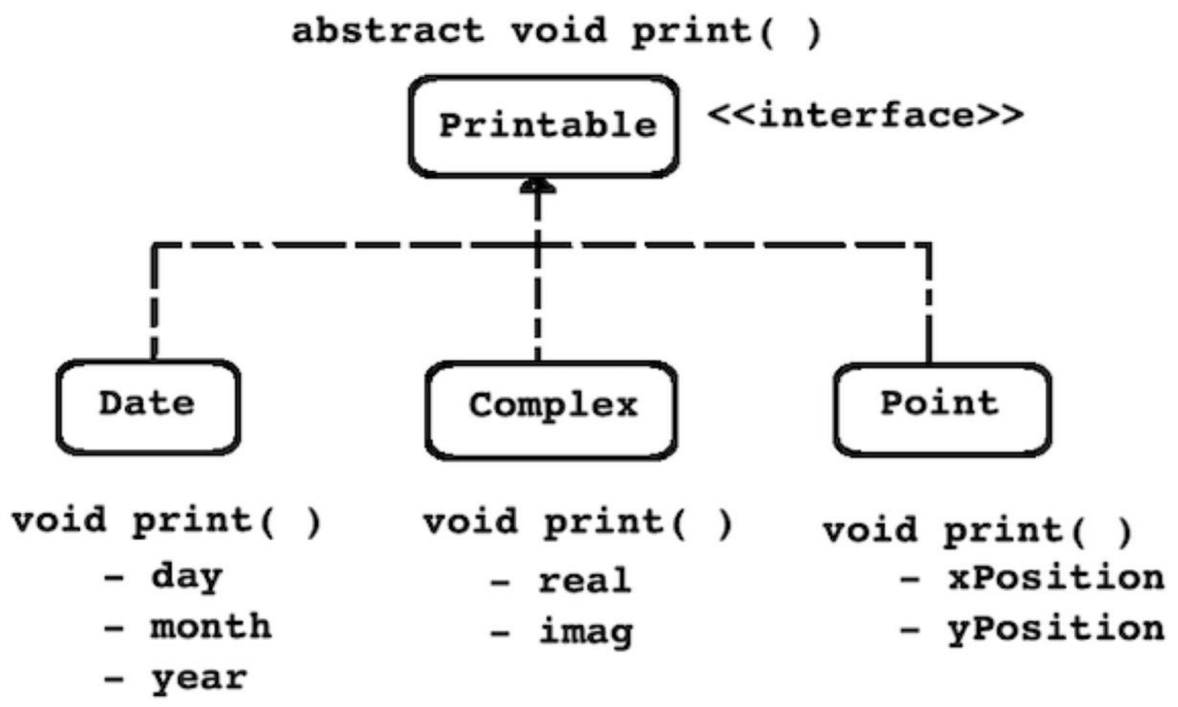
```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle( ); //Upcasting
arr[ 1 ] = new Circle( ); //Upcasting
arr[ 2 ] = new Triangle( ); //Upcasting
```

## Abstract Class

1. If "is-a" relationship is exist between super type and sub type and if we want same method design in all the sub types then super type must be abstract.
2. Using abstract class, we can group instances of related type together
3. Abstract class can extend only one abstract/concrete class.
4. We can define constructor inside abstract class.
5. Abstract class may or may not contain abstract method.

- **Hint** : In case of inheritance if state is involved in super type then it should be abstract.

# Interface



```
Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Date( ); //Upcasting
arr[ 1 ] = new Complex( ); //Upcasting
arr[ 2 ] = new Point( ); //Upcasting
```

# Interface

1. If "is-a" relationship is not exist between super type and sub type and if we want same method design in all the sub types then super type must be interface.

2. Using interface, we can group instances of unrelated type together.

3. Interface can extend more than one interfaces.

4. We can not define constructor inside interface.

5. By default methods of interface are abstract.

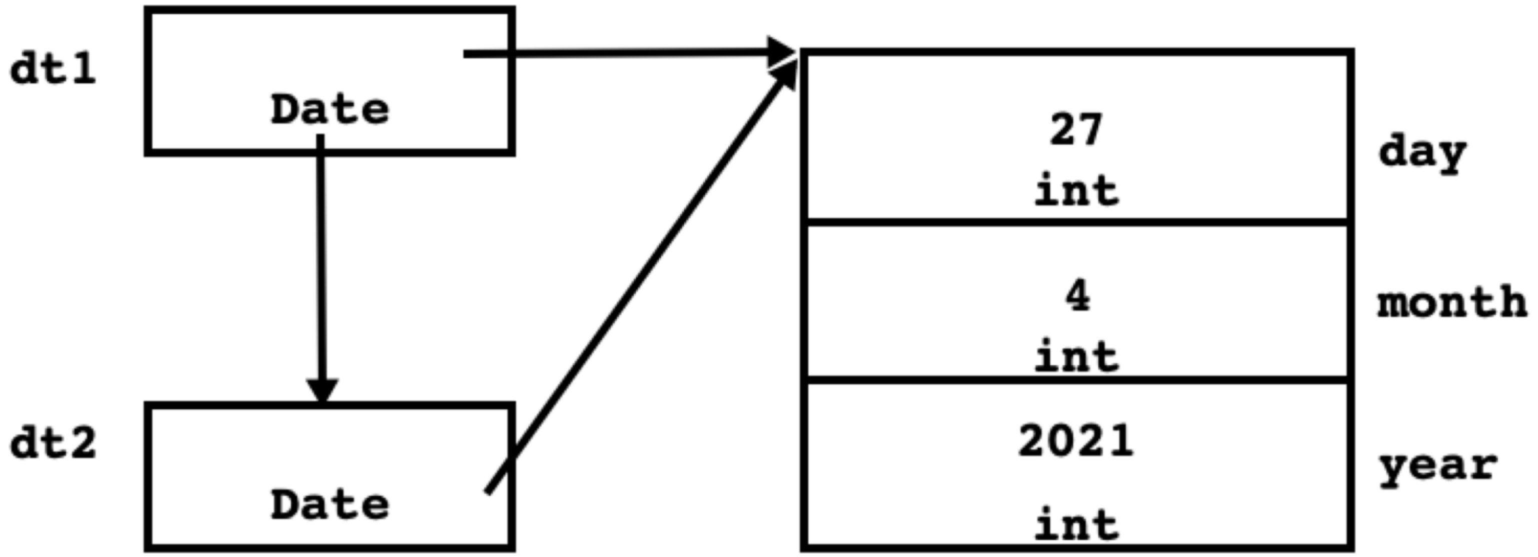
- **Hint** : In case of inheritance if state is not involved in super type then it should be interface.

## Commonly Used Interfaces

1. java.lang.AutoCloseable
2. java.io.Closeable
3. java.lang.Cloneable
4. java.lang.Comparable
5. java.util.Comparator
6. java.lang.Iterable
7. java.util.Iterator
8. java.io.Serializable

## Cloneable Interface Implementation

- `Date dt1 = new Date( 27, 4, 2021 );`
- `Date dt2 = dt1;  //Shallow Copy Of References`

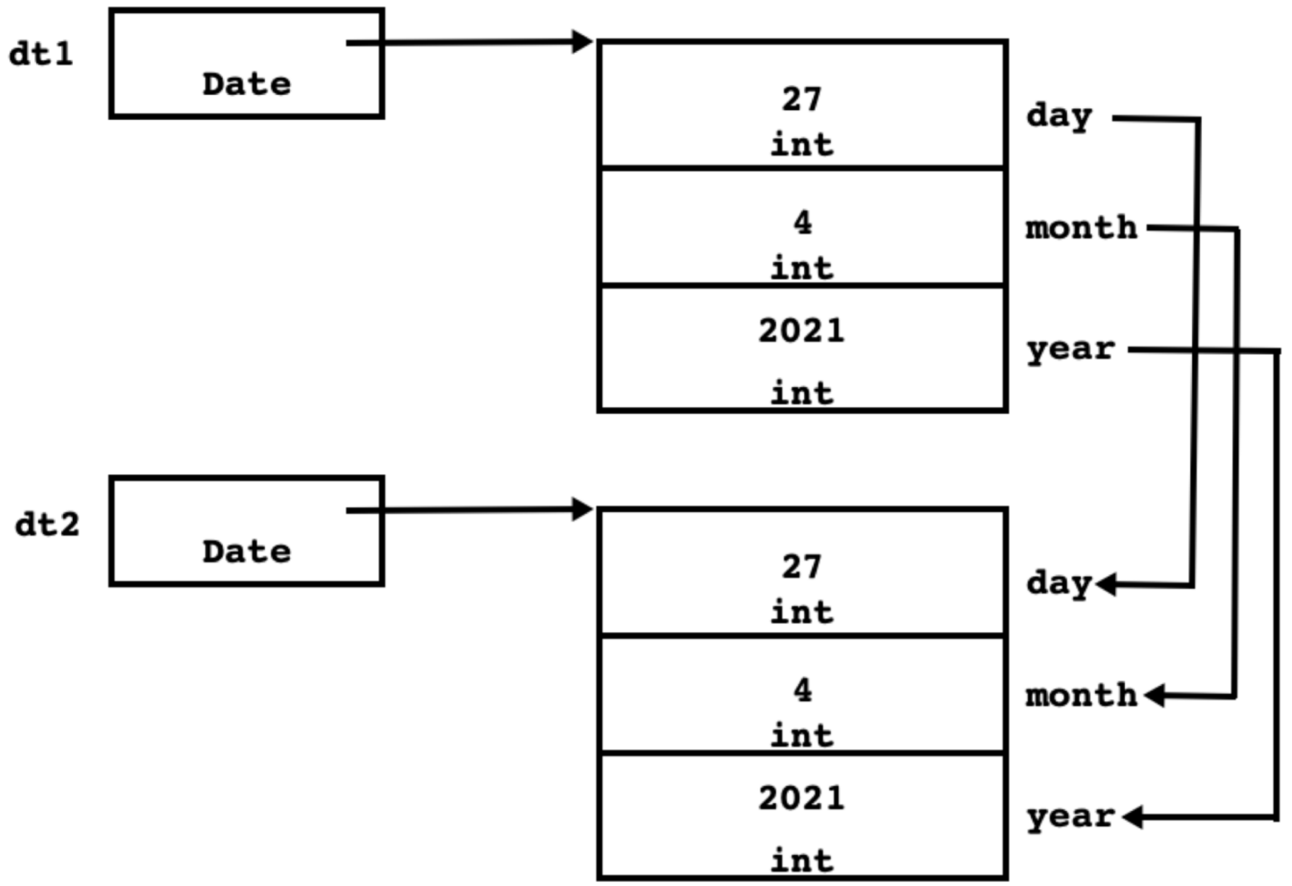


## Cloneable Interface Implementation

- If we want to create new instance from existing instance then we should use clone method.
- clone( ) is non final native method of java.lang.Object class.
- Syntax:
  - **protected native Object clone( ) throws CloneNotSupportedException**
- Inside clone() method, if we want to create shallow copy instance then we should use super.clone( ) method.
- Cloneable is interface declared in java.lang package.
- Without implementing Cloneable interface, if we try to create clone of the instance then clone() method throws CloneNotSupportedException.

# Cloneable Interface Implementation

- `Date dt1 = new Date( 27, 4, 2021 );`
- `Date dt2 = dt1.clone( ); //Shallow Copy Of Instance`



## Marker Interface

- An interface which do not contain any member is called marker interface. In other words, empty interface is called as marker interface.
- Marker interface is also called as tagging interface.
- If we implement marker interface then Java compiler generates metadata for the JVM, which help JVM to clone/serialize or marshal state of object.
- Example:
    1. java.lang.Cloneable
    2. java.util.EventListener
    3. java.util.RandomAccess
    4. java.io.Serializable
    5. java.rmi.Remote

## Comparable

- It is interface declared in java.lang package.
- **"int compareTo(T other)"** is a method of java.lang.Comparable interface.
- If state of current object is less than state of other object then compareTo() method should return negative integer( -1 ).
- If state of current object is greater than state of other object then compareTo() method should return positive integer( +1 ).
- If state of current object is equal to state of other object then compareTo() method should return zero( 0 ).
- If we want to sort, array of non primitive type which contains all the instances of same type then we should implement Comparable interface.

## Comparator

- It is interface declared in java.util package.
- **"int compare(T o1, T o2)"** is a method of java.util.Comparator interface.
- If state of current object is less than state of other object then compare() method should return negative integer( -1 ).
- If state of current object is greater than state of other object then compare() method should return positive integer( +1 ).
- If state of current object is equal to state of other object then compare() method should return zero( 0 ).
- If we want to sort, array of instances of non primitive of different type then we should implement Comparator interface.

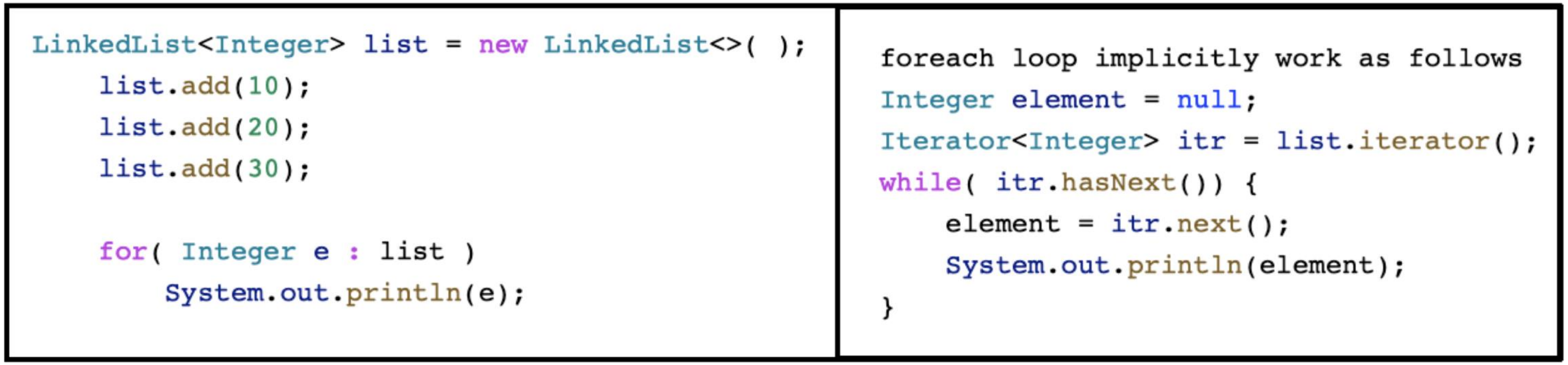
## Iterable and Iterator Implementation

- Iterable<T> is interface declared in java.lang package.
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- It is introduced in JDK 1.5
- Methods of java.lang.Iterable interface:
    1. Iterator<T> iterator()
    2. default Spliterator<T> spliterator()
    3. default void forEach(Consumer<? super T> action)

## Iterable and Iterator Implementation

- Iterator<E> is interface declared in java.util package.
- It is used to traverse collection in forward direction only.
- It is introduced in JDK 1.2
- Methods of java.util.Iterator interface:
    1. boolean hasNext()
    2. E next()
    3. default void remove()
    4. default void forEachRemaining(Consumer<? super E> action)

# Iterable and Iterator Implementation



```
LinkedList<Integer> list = new LinkedList<>( );
    list.add(10);
    list.add(20);
    list.add(30);

    for( Integer e : list )
        System.out.println(e);
```

```
foreach loop implicitly work as follows
Integer element = null;
Iterator<Integer> itr = list.iterator();
while( itr.hasNext()) {
    element = itr.next();
    System.out.println(element);
}
```

Thank You.

akshita.chanchlani@sunbeaminfo.com